Notes for CS 201: Computer System Organization

Based on class and student notes.

December 11, 2023

Contents

1 Main Numeral Systems

- **Binary to Decimal:**

$$(1100\ 0100)_2 = 2^7 + 2^6 + 2^2 = (196)_d$$

- **Binary to Hexadecimal:**

$$(1100\ 0100)_2 \ \rightarrow\ (1100)_2 = (\mathrm{C})_h,\ (0100)_2 = (4)_h \ \rightarrow\ (\mathrm{C4})_h$$

- **Hexadecimal to Decimal:**

$$(\mathrm{C4})_h \ \rightarrow\ (\mathrm{C})_h = (12)_d,\ (4)_h = (4)_d \ \rightarrow\ 12 * 16^1 + 4 * 16^0 = (196)_d$$

- **Hexadecimal to Binary:**

$$(\mathrm{C4})_h \ \rightarrow\ (\mathrm{C})_h = (1100)_2,\ (4)_h = (0100)_2 \ \rightarrow\ (1100\ 0100)_2$$

- **Decimal to Binary:**

$$(196)_d = 2^7 + 2^6 + 2^2 = (1100\ 0100)_2$$

- **Decimal to Hexadecimal:**

$$(196)_d = 2^7 + 2^6 + 2^2 = (1100\ 0100)_2 = (\mathrm{C4})_h$$

You will find that it is super convenient to convert between Binary and Hex. Thus in systems we utilize this feature to store large numbers. And as you may find out, converting by hand does not have a fixed routine. As long as the result is correct, we can apply different methods. Here is a reference table:

Hex	**Dec**	**Bin**
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

2 Data Representation I: Texts and Whole Numbers

2.1 Symbols and Texts

For symbols and texts, we have a standard for the conversion. The most widely adopted version is the ASCII code: (typos may occur due to Latex formatting issue)

Char	Binary	Char	Binary	Char	Binary	Char	Binary
00	00000000		00100000	@	01000000	‘	01100000
01	00000001	!	00100001	A	01000001	a	01100001
02	00000010	”	00100010	B	01000010	b	01100010
03	00000011	#	00100011	C	01000011	c	01100011
04	00000100	$	00100100	D	01000100	d	01100100
05	00000101	%	00100101	E	01000101	e	01100101
06	00000110	&	00100110	F	01000110	f	01100110
07	00000111	’	00100111	G	01000111	g	01100111
08	00001000	(	00101000	H	01001000	h	01101000
tab	00001001	)	00101001	I	01001001	i	01101001
enter	00001010	*	00101010	J	01001010	j	01101010
0b	00001011	+	00101011	K	01001011	k	01101011
0c	00001100	,	00101100	L	01001100	l	01101100
r	00001101	-	00101101	M	01001101	m	01101101
0e	00001110	.	00101110	N	01001110	n	01101110
0f	00001111	/	00101111	O	01001111	o	01101111
10	00010000	0	00110000	P	01010000	p	01110000
11	00010001	1	00110001	Q	01010001	q	01110001
12	00010010	2	00110010	R	01010010	r	01110010
13	00010011	3	00110011	S	01010011	s	01110011
14	00010100	4	00110100	T	01010100	t	01110100
15	00010101	5	00110101	U	01010101	u	01110101
16	00010110	6	00110110	V	01010110	v	01110110
17	00010111	7	00110111	W	01010111	w	01110111
18	00011000	8	00111000	X	01011000	x	01111000
19	00011001	9	00111001	Y	01011001	y	01111001
1a	00011010	:	00111010	Z	01011010	z	01111010
1b	00011011	;	00111011	[	01011011	{	01111011
1c	00011100	¡	00111100	\	01011100	—	01111100
1d	00011101	=	00111101	]	01011101	}	01111101
1e	00011110	¿	00111110	ˆ	01011110	˜	01111110
1f	00011111	?	00111111	_	01011111	7f	01111111

2.2 Integer Types

There are several data types that can represent integer: short, int, long. But when it comes to positive/negative numbers, the binary system works a bit tricky here.

- Unsigned: We can use usual binary representation for unsigned integers.
- Signed: For signed integers, we will apply **2's Complement** to represent them.

Algorithm for 2's Complement ($d \to b$):

1. Convert it to binary format;
2. If the decimal is positive, we are done;
3. If the decimal is negative, flip the digits and add 1.

Algorithm for 2's Complement ($b \to d$):

1. Check the first digit of the binary;
2. If the 1st digit is 0, the decimal is positive. Convert to decimal;
3. If the 1st digit is 1, the decimal is negative. Flip and add 1. Convert to decimal.

2.3 Floating Points

For a floating point number, its standard binary representation is composed of three parts: **sign**, **exponent**, and **fraction**. Here is an example of the algorithm:

$$\begin{aligned} 5.75 \rightarrow 101.11 &\rightarrow 1.0111 * 2^2 \\ sign &= 0 \\ exponent &= 2 + 127 = 129 = (10000001)_b \\ fraction &= 0111...000 \end{aligned}$$

Algorithm for converting from Float to Binary:

1. Confirm the sign: positive→0, negative→1;
2. Convert to binary I: for the whole number, directly convert to binary;
3. Convert to binary II: for the fraction, multiply by 2 and pop the whole number part(1 or 0) until we get zero or loop;
4. Get the exponent: convert the form "$xxxx.xxx$" to "$1.xxx * 2^n$", then convert (n+127) to binary, this is the exponent;
5. Get the fraction: put everything behind the dot to the fraction part.

3 Data Representation II: IEEE Floating Point

3.1 Reserved Values

The above format of turning floating point number to binary code is also called **IEEE 754** method. There are also some extreme values and that are reserved in special format:

1. **Infinity**(∞): Exponent value = 1111 1111, Mantissa = all zeros;
2. **Not a Number**(NAN): Exponent value = 1111 1111, Mantissa = not all zeros;
3. **Zero**(0): Exponent value = 0000 0000, Mantissa = all zeros;
4. **Subnormal**(very small): Exponent value = 0000 0000, Mantissa = not all zeros;

3.2 Ranges

Here is the range for the binary represented floating point numbers.

IEEE 754 Format	sign	Exponent	Mantissa	Exponent Bias
32 bit single precision	1 bit	8 bits	23 bits (+1 not stored)	$2^{(8-1)} - 1 = 127$
64 bit double precision	1 bit	11 bits	52 bits (+1 not stored)	$2^{(11-1)} - 1 = 1023$
128 bit quadruple precision	1 bit	15 bits	112 bits (+1 not stored)	$2^{(15-1)} - 1 = 16383$

4 Data Representation III: Sounds and Videos

4.1 Types of Information

- **Sounds:** As sounds are captured by sound cards, they are converted into binary code. There are two keywords here: "sampling rate" and "quantization". Sampling is to capture the sound at certain rate(Hz). The rate here means how much "captures" happen per second. Quantization means to convert the captured thing into some kind of binary code according to certain standard.
- **Images:** The basic unit of image is "pixel". Each pixel can be represented in binary code differently under different standards.
  1. Black/White: each pixel is 0(black) or 1(white);
  2. Gray: each pixel is 1 set of digits to represent gray shade;
  3. RGB: each pixel is 3 sets of digits;
  4. CMYK: each pixel is 4 sets of digits.
- **Videos:** can be seen as sets of images.

4.2 Sizes

By the end of Data Representation, we will discuss about the "sizes":
Powers of two:

$2^0 = 1$	$2^5 = 32$	$2^{10} = 1024$ =1K(kilo)
$2^1 = 2$	$2^6 = 64$	2^{20} =1M(Mega)
$2^2 = 4$	$2^7 = 128$	2^{30} =1K(Giga)
$2^3 = 8$	$2^8 = 256$	2^{40} =1T(Tera)
$2^4 = 16$	$2^9 = 512$	2^{50} =1P(Peta)

Rule: $2^{n+m} = 2^n * 2^m$
Example: $2^{16} = 2^{10} * 2^6 = 1k * 64 = 64k$

Logarithms (base 2): (Reverse of Power of two Table)
Rule: $\log_2(n * m) = \log_2 n + \log_2 m$
Example: $\log_2 64k = \log_2 1k + \log_2 64 = 10 + 6 = 16$

*CPU and Memory are connect by: control bus, data bus, and address bus.

5 Intro to C Programming Language

5.1 Pointers in C

A Pointer in C language is a variable that stores an address. See the simple code below:

```
#include <stdio.h>

int main() {

    int x;      //Declare an integer variable x.
    int *p;     //Declare an int pointer variable.
                //(ie. the type of p is "int")
    p = &x;     //Assign the address of lowest byte x to p
    *p = 25;    //Assign 25 to where p points to (*p is also called the de-reference operator)
    int y=*p;   //Assign 25 to y (equivalent to: int y=x)
    int *q=p;   //Create another pointer that points to what p points to

    printf("Hello, LaTeX!\n");
    return 0;
}
```

There are several things we need to know:

1. The size of a pointer is 32 or 64 (based on the type of CPU), which allows access to 2^{32} or 2^{64} different addresses;
2. "%p" is used in printing statements to show the address of a pointer in hexadecimal format;
3. When a program tries to access a byte memory that's not allowed to, the program will crash.

5.2 Pointers to Array

A pointer can point to array and have multiple behaviors. See the example code below:

```
#include <stdio.h>

int main() {
    int a[3] = {1, 2, 3};   //Initialize an array of integers
    int *p = a;             //Set p points to address of first element of a
                            //(equivalent to p=&a[0])
    p++;                    //Move pointer to the next element of a
                            //Increase address by appropriate # of bytes
    p--;                    //Move pointer to the previous element of a
                            //Decrease address by appropriate # of bytes
    (*p)++;                 //Increment what p points to

    printf("Hello, LaTeX!\n");
    return 0;
}
```

5.3 Application Memory

- There are different sections for any program:
    1. Stack: when function is called, all its local variables will reside in the stack (pushed into the stack);
    2. Heap: when data are allocated using malloc, they will reside in heap and need to be freed manually in C;
    3. Global/Static: all global/static variables and functions will be stored in this area;
    4. Code: codes that are to be executed are stored in this area.

6 Memory Allocation

In C and C++, memory allocation functions like 'malloc', 'calloc', 'realloc', and 'free' are used to manage dynamic memory allocation. Here's a brief overview of each of these functions:

1. `malloc` (Memory Allocation):
   - `void *malloc(size_t size);`
   - `malloc` stands for "memory allocation."
   - It allocates a block of memory of the specified size in bytes on the heap (dynamically allocated memory).
   - It returns a pointer to the first byte of the allocated memory block.
   - The memory allocated by `malloc` is uninitialized, which means it contains garbage values.
   - It does not initialize the memory to zero.
2. `calloc` (Contiguous Allocation):
   - `void *calloc(size_t num_elements, size_t element_size);`
   - `calloc` stands for "contiguous allocation."
   - It allocates a block of memory for an array of elements, each of the specified size, and initializes the memory to zero.
   - It returns a pointer to the first byte of the allocated memory block.
3. `realloc` (Reallocate Memory):
   - `void *realloc(void *ptr, size_t new_size);`
   - `realloc` is used to change the size of a previously allocated memory block.
   - It takes an existing pointer `ptr` to a dynamically allocated memory block and resizes it to the new size.
   - It can be used to make the memory block larger or smaller.
   - It may move the memory block to a new location if resizing is not possible in place.
   - The contents of the old block are preserved up to the minimum of the old and new sizes.
4. `free` (Free Memory):
   - `void free(void *ptr);`
   - `free` is used to deallocate (release) memory previously allocated by `malloc`, `calloc`, or `realloc`.
   - After calling `free`, the memory can be reused for other purposes.
   - It's important to free dynamically allocated memory to prevent memory leaks.

Here is the sample code:

```
int *p;                          //Declear a pointer p
p = malloc(10* sizeof(int));     //Assign the address 1st int of the 10 ints to p
free(p);                         //De-allocate p
p = NULL;                        //Re-allocate p to NULL for convenience
```

6.1 Dynamic Arrays

Arrays can be static arrays (stored in the stack) or dynamic arrays (created using malloc and stored in heap) with fixed sizes (can be modified). Here is a sample code:

```
int a[10] = {0};                          //static array with fixed length

int *b = (int*)malloc(sizeof(int)*10);    //dynamic array with fixed length
```

Here is another example of Dynamic Array:

```
int *a;
a = (int*) malloc (3*sizeof(int));
*a = 10;        // equivalent to a[0] = 10;
*(a+1) = 20;    // equivalent to a[1] = 20;
```

Here is another example, array of pointers:

```
int* a[3];
int x = 5;
a[0] = &x;
a[1] = (int*) malloc(sizeof(int)*10);
*a[1] = 10;
a[2] = (int*) malloc(3*sizeof(int));

printf("%d\n", *a[0]);  // => 5
printf("%d\n", *a[1]);  // => 10
printf("%d\n", *a[2]);  // => 0 (should be garbage value, but may vary due to OS)

free(a[1]); a[1]=NULL;
free(a[2]); a[2]=NULL;
```

6.2 Strings and Pointers

Note that there is no such thing like String in C. Instead, we use an array of characters that ends with "\0" to represent String. Here is an example:

```
char c[10] = "welcome";
// equivalent to:
char c[10] = {'w', 'e', 'l', 'c', 'o', 'm', 'e', '\0'};
```

We can also utilize pointers when dealing with char arrays. Here is an example:

```
char c[10] = "welcome";
char *p = c;
printf("%s", p);        //print "welcome"
printf("%c", *p);       //print "w"
```

We can also copy char array using some techniques. Here is an example:

```
char S_1[10] = "welcome";
char s2[10];
char *p, *q;
p = S_1;
q = s2;
while((*q++ = *p++));
```

6.3 Structs and Linked-Lists

In this section, we'll explore how to use C structs to implement a simple singly linked-list data structure. Linked-lists are a fundamental data structure in computer science and are used to store and manage collections of data.

6.3.1 Defining a Structs for Linked-List Node

To create a linked-list, we'll first define a struct to represent a node in the list. Each node contains two parts: the data it holds and a pointer to the next node in the list (or `NULL` if it's the last node). Here's how we define a struct for a linked-list node:

```
deftype struct Node {
    int data;
    struct Node* next;
}NOTE;
```

In this example, we've defined a struct named `Node` with two members: `data`, an integer to store the node's data, and `next`, a pointer to the next node in the list.

6.3.2 Creating a Linked-List

Now that we have our node struct, we can create a linked-list by creating instances of these nodes and linking them together. Here's an example of how to create a simple linked-list with three nodes:

```
NODE head = NULL; // Initialize an empty linked-list

// Create nodes and assign data
struct Node* node1 = (struct Node*)malloc(sizeof(struct Node));
node1->data = 10;
node1->next = NULL;

struct Node* node2 = (struct Node*)malloc(sizeof(struct Node));
node2->data = 20;
node2->next = NULL;
...

// Link nodes to form the list
head = node1;
node1->next = node2;
...
```

In this code, we first initialize an empty linked-list by setting the `head` pointer to `NULL`. Then, we create three nodes (`node1`, `node2`, and `node3`), assign data to each node, and link them together to form a simple linked-list.

6.3.3 Traversing the Linked-List

To access the data in a linked-list, we need to traverse the list from the `head` node to the end. Here's how you can traverse and print the data in our example linked-list:

```
struct Node* current = head; // Start at the head of the list

while (current != NULL) {
    printf("%d -> ", current->data);
    current = current->next; // Move to the next node
}

printf("NULL\n"); // End of the list
```

This code initializes a `current` pointer to the `head` of the list and iterates through the list, printing the data in each node. The loop terminates when `current` becomes `NULL`, indicating the end of the list.

6.3.4 Adding to the Linked-List

We can also add to the LinkedList. Here is the following sample code:

```
// Function to add a node to the front of the linked-list
void addToFront(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *head;
    *head = newNode;
}

// Function to add a node to the end of the linked-list
void addToEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
    }
}
```

6.3.5 Delete in the Linked-List

We can also delete some node in the LinkedList. Here is the following sample code:

```
    // Function to delete a node with the given data from the linked-list
void deleteNode(struct Node** head, int data) {
    struct Node* current = *head;
    struct Node* prev = NULL;

    // Traverse the list to find the node to delete
    while (current != NULL && current->data != data) {
        prev = current;
        current = current->next;
    }

    // If the node to delete is found
    if (current != NULL) {
        // If it's the first node
        if (prev == NULL) {
            *head = current->next;
        } else {
            prev->next = current->next;
        }
        free(current); // Free the memory of the deleted node
    }
}
```

6.3.6 Conclusion

Structs in C provide a powerful way to define custom data types, making them ideal for implementing data structures like linked-lists. By creating a struct to represent a node and using pointers to link nodes together, we can create and manipulate linked-lists to store and manage data efficiently.

7 Bit-wise Operators in C

There are several bit-wise operators in C. They are different from the logical operators and serve for unique but useful purposes.

7.1 Operators

- **∼:** NOT operator → 1 to 0, 0 to 1
  1. char x = 7 → 0000 0111
     ∼x → 1111 1000 → -8
  2. char x = 4 → 0000 0100
     ∼x → 1111 1011 → -5
  3. Note that x+(∼x) will be -1 if it is signed int.
- **&:** AND operator → 1 if both 1, otherwise 0
  1. char x = 7 → 0000 0111
     char y = 4 → 0000 0100
     x & y → 0000 0100 → 4
  2. char x = -7 → 1111 1001
     char y = -4 → 1111 1100
     x & y → 1111 1000 → -8
- **|:** OR operator → 0 if both 0, otherwise 1
  1. char x = 7 → 0000 0111
     char y = 4 → 0000 0100
     x | y → 0000 0111 → 7
- **ˆ:** XOR operator → 0 is same, 1 if different
  1. char x = 7 → 0000 0111
     char y = 4 → 0000 0100
     x ˆ y → 0000 0011 → 3
- **≪:** LEFT SHIFT operator → move left, fill with 0
  1. char x = 7 → 0000 0111;
     x ≪ 2; → 0001 1100 → 28
  2. char x = -4 → 1111 1100;
     x ≪ 2; → 1111 0000 → -16
- **≫:** RIGHT SHIFT operator → move right, fill with neighbor
  1. char x = 7 → 0000 0111;
     x ≫ 2; → 0000 0001 → 1
  2. char x = -4 → 1111 1100;
     x ≫ 2; → 1111 1111 → -1

7.2 Examples of Using Operators

In this section, we'll explore various examples of using bitwise operators in C for different purposes.

7.2.1 Creation of Content

Bitwise operators can be used to create specific bit patterns for various purposes. Here are some common examples:

1. **All Zeros**:

   ```
   int allZeros = 0;
   ```

   All bits in the variable `allZeros` are set to 0.

2. **All Ones**:

   ```
   int allOnes = -1; // or int allOnes = ~0;
   ```

   All bits in the variable `allOnes` are set to 1.

3. **1 at a Specific Location**: To set a single bit at a specific location (e.g., bit 3), you can use bitwise OR (`|`) with a mask:

   ```
   int value = 0;
   int mask = 1 << 3; // Set bit 3
   value |= mask;
   ```

   The variable `value` now has bit 3 set to 1.

4. **K Bits with 1 at Certain Locations**: To set multiple bits to 1 at specific locations (e.g., bits 0, 2, and 5), you can use bitwise OR (`|`) with masks:

   ```
   int value = 0;
   int mask1 = 1 << 0; // Set bit 0
   int mask2 = 1 << 2; // Set bit 2
   int mask3 = 1 << 5; // Set bit 5
   value |= mask1 | mask2 | mask3;
   ```

   The variable `value` now has bits 0, 2, and 5 set to 1.

5. **K Bits with 0 at Certain Locations**: To set multiple bits to 0 at specific locations (e.g., bits 1, 4, and 6), you can use bitwise AND (`&`) with inverted masks:

   ```
   int value = -1; // All ones
   int mask1 = ~(1 << 1); // Clear bit 1
   int mask2 = ~(1 << 4); // Clear bit 4
   int mask3 = ~(1 << 6); // Clear bit 6
   value &= mask1 & mask2 & mask3;
   ```

   The variable `value` now has bits 1, 4, and 6 set to 0.

7.2.2 Verification

Bitwise operators can also be used for verification purposes, such as checking if specific bits are set or cleared in a value. Here's an example using the bitwise AND operator (`&`) for verification:

```
#include <stdio.h>

int main() {
    int value = 0B_10101010; // Binary value: 10101010
    int mask = 0B_00101010;  // Binary mask: 00101010

    if ((value & mask) == mask) {
        printf("Mask is set in the value.\n");
    } else {
        printf("Mask is not set in the value.\n");
    }

    return 0;
}
```

In this code, we use the bitwise AND operator to check if a specific mask is set in a value. There are also some other simple applications:

1. Check if i^{th} bit in x is 1 or 0 $\rightarrow$ x & ($1 \ll i$);
2. Set the i^{th} bit to 1 $\rightarrow$ x | ($1 \ll i$);
3. Clear the i^{th} bit (set to 0) $\rightarrow$ x & $\sim(1 \ll i)$.

7.2.3 Faster Algorithm

Bitwise operators are often used to implement faster algorithms by optimizing certain operations. For example, you can use bitwise left shift and right shift operators to perform multiplication and division by powers of two efficiently. Here's an example:

```
#include <stdio.h>

int main() {
    int num = 10;
    int result;

    // Multiply num by 4 using left shift
    result = num << 2; // Equivalent to num * 4
    printf("Result of multiplication: %d\n", result);

    // Divide num by 8 using right shift
    result = num >> 3; // Equivalent to num / 8
    printf("Result of division: %d\n", result);

    return 0;
}
```

In this code, we use bitwise left shift and right shift to efficiently perform multiplication and division by powers of two, which can be significantly faster in some cases.

These examples demonstrate how bitwise operators can be applied in various scenarios for content creation, verification, and algorithm optimization.

8 Intro to Assembly Language

8.1 Basics

mov: store values

```
mov $10, %rax   //rax = 10
mov %rax, %rbx  //rbx = rax
```

mov: absolute address

```
mov 0xAB_0F, %rax    //move the content at 0xAD0F into rax
```

mov: de-reference pointer

```
mov (%rcx), %rax    //Load the content at rcx
                    //Move the content into rax
                    //rax = *rcx
mov %rcx, (%rax)    //Move rcx into the content of rax
                    //*rax = rcx
```

mov: use offset from pointer

```
mov 8(%rcx), %rax   //rax = *(rcx+8)
```

mov: index addressing

```
mov (%rsi, %rcx, 4), %rax   //rax = *(rsi + [rcx*4])
```

- rsi: base
- rcx: index
- 4: size of each element

mov: index with offset

```
mov 20(%rsi, %rcx, 4), %rax   //rax = *(20 + rsi + [rcx*4])
```

- 20: offset
- rsi: base
- rcx: index
- 4: size of each element

Sample Code

```
struct grades{
    int count;
    int a[10];
    float avg;
}

struct grades *p;
p = malloc(sizeof(struct grades);
p -> a[3] = 500;
// eqv to: mov $500, 4(%rsi, %rcx, 4)
```

8.2 Registers

When we are talking about registers, we mainly focus on passing arguments, or the special registers (and rules) that we use when dealing with functions. Thus we will start with functions.

8.2.1 return value

There is only one rule here: Returned value must be in the **%rax** register.

8.2.2 pass arguments (parameters)

1. First six: rdi, rsi, rdx, rcx, r8, r9
2. Others: pushed to the stack, right to left

Example

```
int find_sum(int x1, x2, x3, x4, x5, x6, x7, x8){
    return x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8;
}

void g(){
    int z;
    z = find_sum(10, 20, 30, 40, 50, 60, 70, 80);
}
```

In assembly, it is represented as:

```
g:              |   find_sum:
    ----        |       ----
    ----        |       ----
mov $10, %rdi   |   mov $0, %rax
mov $20, %rsi   |   add %rdi, %rax
mov $30, %rdx   |   add %rsi, %rax
    ----        |       ----
mov $60, %r9    |   add %r9, %rax
push $80        |   pop %rbx
push $70        |   add %rbx, %rax
call find_sum   |   pop %rbx
    ----        |   add %rbx, %rax
                |       ----
```

8.2.3 Saved Register

1. **caller-saved:** rax, rcx, rdx, rdi, rsi, rsp, r8, r9, r10, r11
2. **callee-saved:** rbx, rbp, r12, r13, r14, r15

Note: 'saved' here means that those registers should not be used and are reserved for the other function.

8.2.4 Using the Stack

In assembly language, 'RSP' (Stack Pointer) and 'RBP' (Base Pointer) are two very important registers, especially in 64-bit architectures (like x86-64). These registers are primarily used for managing function calls and stack operations.

- **RSP (Stack Pointer)**
  - **Function**
    'RSP' is the stack pointer register. It always points to the current top of the stack. In 64-bit architectures, it is referred to as 'RSP', and in 32-bit architectures, it is 'ESP'.

- **Usage**
  1. During function calls, arguments are typically pushed onto the stack, and then 'RSP' is adjusted to point to the new top of the stack.
  2. Upon function return, 'RSP' is reset to point to the position before the function call, ensuring the correct restoration of the caller's stack frame.
  3. 'RSP' is usually managed automatically by the operating system and the compiler, but in low-level operations, such as manual stack management or assembly programming, programmers might need to manipulate it directly.

- **RBP (Base Pointer)**
  - **Function**
    'RBP' is the base pointer register, used to point to the base or bottom of a stack frame. In 64-bit architectures, it is referred to as 'RBP', and in 32-bit architectures, it is 'EBP'.
  - **Usage**
    1. At the start of a function call, 'RBP' is typically set to the current value of 'RSP', making 'RBP' point to the start of the stack frame.
    2. 'RBP' can be used to access function parameters and local variables, achieved by offsets relative to 'RBP'. This is beneficial as 'RBP' remains constant regardless of the changes to the stack top ('RSP'), thus stably referencing parameters and local variables.
    3. At the end of a function, 'RBP' is usually restored to its value before the function call to maintain the integrity of the stack.

Example

```
my_function:
    // Function prologue
    push rbp         // Save the old base pointer value
    mov rbp, rsp     // Set the new base pointer

    // Function body
    // Perform function tasks here
    // ...

    // Function epilogue
    mov rsp, rbp     // Restore the stack pointer
    pop rbp          // Restore the old base pointer value
    ret              // Return from the function
```

8.3 Operations and Functions

Before we dive into functions, we want to highlight that assembly code only do simple stuffs. We can do very limited operations in one line of assembly code.

General Assembly Operations

1. Single simple arithmetic operations
2. Transfer data between memory and register
   (a) Load data from memory to register
   (b) Store data register into memory
3. Transfer control
   (a) Unconditional jumps to/from procedures
   (b) Conditional branches

8.3.1 Arithmetic Instructions

(see Table 1 to 4 on page 20)

8.3.2 Conditional Control and Loops

(see Table 5 to 7 on page 20)

Sample Codes

- Note that for 32-bit system, we use `eax, edi, esi, ...` instead of `rax, rdi, rsi, ...` that we used in 64-bit system;
- Note that following codes table page comes from textbook *Dive Into Systems*.

1. **getSmallest:** getting familiar with basic conditional statements

```
//c version
int getSmallest(int x, int y) {
int smallest;
if ( x > y ) {
    smallest = y;
}
else {
    smallest = x;
}
return smallest;
}
//assmbly version
0x40059a <+4>:  mov %edi,-0x14(%rbp)          # copy x to %rbp-0x14
0x40059d <+7>:  mov %esi,-0x18(%rbp)          # copy y to %rbp-0x18
0x4005A_0 <+10>: mov -0x14(%rbp),%eax          # copy x to %eax
0x4005a3 <+13>: cmp -0x18(%rbp),%eax          # compare x with y
0x4005a6 <+16>: jle 0x4005B_0 <getSmallest+26> # if x<=y goto <getSmallest+26>
0x4005a8 <+18>: mov -0x18(%rbp),%eax          # copy y to %eax
0x4005ae <+24>: jmp 0x4005b9 <getSmallest+35> # goto <getSmallest+35>
0x4005B_0 <+26>: mov -0x14(%rbp),%eax          # copy x to %eax
0x4005b9 <+35>: pop %rbp                      # restore %rbp (clean up stack)
0x4005ba <+36>: retq                          # exit function (return %eax)
```

2. **getSmallest_cmov:** using cmov to create more efficient code

```
//c version
int getSmallest_cmov(int x, int y) {
    return x > y ? y : x;
}
//assembly version
0x4005d7 <+0>:   push   %rbp             #save %rbp
0x4005d8 <+1>:   mov    %rsp,%rbp        #update %rbp
0x4005db <+4>:   mov    %edi,-0x4(%rbp)  #copy x to %rbp-0x4
0x4005de <+7>:   mov    %esi,-0x8(%rbp)  #copy y to %rbp-0x8
0x4005e1 <+10>:  mov    -0x8(%rbp),%eax  #copy y to %eax
0x4005e4 <+13>:  cmp    %eax,-0x4(%rbp)  #compare x and y
0x4005e7 <+16>:  cmovle -0x4(%rbp),%eax  #if (x <=y) copy x to %eax
0x4005eb <+20>:  pop    %rbp             #restore %rbp
0x4005ec <+21>:  retq                    #return %eax
```

3. **sumUp: (while loop)**

```
//c version
int sumUp(int n){
int total, i = 0, 1;
while (i <= n) {
    total += i;
    i += 1;
}
return total;
}
//assembly version
0x400526 <+0>:  push %rbp              # save %rbp onto the stack
0x400527 <+1>:  mov  %rsp,%rbp         # update the value of %rbp (new frame)
0x40052a <+4>:  mov  %edi,-0x14(%rbp)  # copy n to %rbp-0x14
0x40052d <+7>:  mov  $0x0,-0x8(%rbp)   # copy 0 to %rbp-0x8 (total)
0x400534 <+14>: mov  $0x1,-0x4(%rbp)   # copy 1 to %rbp-0x4 (i)
0x40053b <+21>:  jmp    0x400547 <sumUp+33>  # goto <sumUp+33>
0x40053d <+23>:  mov    -0x4(%rbp),%eax      # copy i to %eax
0x400540 <+26>:  add    %eax,-0x8(%rbp)      # add i to total (total += i)
0x400543 <+29>:  add    $0x1,-0x4(%rbp)      # add 1 to i (i += 1)
0x400547 <+33>:  mov    -0x4(%rbp),%eax      # copy i to %eax
0x40054a <+36>:  cmp    -0x14(%rbp),%eax     # compare i to n
0x40054d <+39>:  jle    0x40053d <sumUp+23>  # if (i <= n) goto <sumUp+23>
0x40054f <+41>:  mov    -0x8(%rbp),%eax     # copy total to %eax
0x400552 <+44>:  pop    %rbp                # restore rbp
0x400553 <+45>:  retq                       # return (total)
```

4. **sumUp2: (for loop)**

```
//c version
int sumUp2(int n) {
int total, i = 0, 1;
for (i; i <= n; i++) {
    total += i;
}
return total;
}
//assembly version
0x400554 <+0>:   push   %rbp                    #save %rbp
0x400555 <+1>:   mov    %rsp,%rbp               #update %rpb (new stack frame)
0x400558 <+4>:   mov    %edi,-0x14(%rbp)        #copy %edi to %rbp-0x14 (n)
0x40055b <+7>:   movl   $0x0,-0x8(%rbp)         #copy 0 to %rbp-0x8 (total)
0x400562 <+14>:  movl   $0x1,-0x4(%rbp)         #copy 1 to %rbp-0x4 (i)
0x400569 <+21>:  jmp    0x400575 <sumUp2+33>    #goto <sumUp2+33>
0x40056b <+23>:  mov    -0x4(%rbp),%eax         #copy i to %eax [loop]
0x40056e <+26>:  add    %eax,-0x8(%rbp)         #add i to total (total+=i)
0x400571 <+29>:  addl   $0x1,-0x4(%rbp)         #add 1 to i (i++)
0x400575 <+33>:  mov    -0x4(%rbp),%eax         #copy i to %eax [start]
0x400578 <+36>:  cmp    -0x14(%rbp),%eax        #compare i with n
0x40057b <+39>:  jle    0x40056b <sumUp2+23>    #if (i <= n) goto loop
0x40057d <+41>:  mov    -0x8(%rbp),%eax         #copy total to %eax
0x400580 <+44>:  pop    %rbp                    #prepare to leave the function
0x400581 <+45>:  retq                           #return total
```

Instruction	Translation
`add S, D`	S + D → D
`sub S, D`	D - S → D
`inc D`	D + 1 → D
`dec D`	D - 1 → D
`neg D`	-D → D
`imul S, D`	S × D → D
`idiv S`	%rax / S: quotient → %rax, remainder → %rdx

Table 1: Common Arithmetic Instructions.

Instruction	Translation	Arithmetic or Logical?
`sal v, D`	D << v → D	arithmetic
`shl v, D`	D << v → D	logical
`sar v, D`	D >> v → D	arithmetic
`shr v, D`	D >> v → D	logical

Table 2: Bit Shift Instructions.

Instruction	Translation
`and S, D`	S & D → D
`or S, D`	S | D → D
`xor S, D`	S ∧ D → D
`not D`	∼D → D

Table 3: Bitwise Operations.

Instruction	Translation	Value
`lea 8(%rax), %rax`	8 + %rax → %rax	13 → %rax
`lea (%rax, %rdx), %rax`	%rax + %rdx → %rax	9 → %rax
`lea (,%rax,4), %rax`	%rax × 4 → %rax	20 → %rax
`lea -0x8(%rcx), %rax`	%rcx - 8 → %rax	0x800 → %rax
`lea -0x4(%rcx, %rdx, 2), %rax`	%rcx + %rdx × 2 - 4 → %rax	0x80c → %rax

Table 4: Example `lea` Operations.

Instruction	Translation
`cmp R1, R2`	Compares R2 with R1 (i.e., evaluates R2 - R1)
`test R1, R2`	Computes R1 & R2

Table 5: Conditional Control Instructions.

Instruction	Description
`jmp L`	Jump to location specified by L
`jmp *addr`	Jump to specified address

Table 6: Direct Jump Instructions.

Signed Comparison	Unsigned Comparison	Description
`je (jz)`		`jump if equal (==) or jump if zero`
`jne (jnz)`		`jump if not equal (!=)`
`js`		`jump if negative`
`jns`		`jump if non-negative`
`jg (jnle)`	`ja (jnbe)`	`jump if greater (>`
`jge (jnl)`	`jae (jnb)`	`jump if greater than or equal (>=)`
`jl (jnge)`	`jb (jnae)`	`jump if less (<)`
`jle (jng)`	`jbe (jna)`	`jump if less than or equal (<=)`

Table 7: Conditional Jump Instructions; Synonyms Shown in Parentheses.

9 Cache Memory

9.1 What is Cache Memory?

Cache is a small amount of memory that is on CPU or right next to CPU. It can provide CPU with the same of its speed. Here are some key points you should know about cache:

1. It stores a copy of info. From the main memory;
2. CPU asks the cache if yes(cache hit) if no(cache miss);
3. The greater the cache hits → the greater the performance;
4. The greater the cache misses → the lower the performance.

9.2 Mapping

CPU does not directly grab information from RAM when cache memory is present. However, due to the limitations of cache, we have to use the technique of **Mapping** to operate. The process is like projecting the actual RAM address onto cache using certain rules. (The limitation or process of projection will cause loss and this is where cache hit/miss comes from.) To understand this mapping/projecting process, we first need to look into the **structure of cache**:

1. **index:** the "address" of the cache line, determined by the number of cache blocks we have ⇒ last part of RAM address
2. **tag:** part of the content of cache line, determined by RAM address minus offset and index ⇒ the rest of RAM address
3. **validation:** the digit that tags if the data in this cache line is valid or not
4. **offset:** used to determine the actual position inside one RAM block, determined by block size

Now let's utilize this with some sample exercises with calculation:

Exercise 1

Consider a 64 Blocks cache and a block size of 16 bytes and address length is 16 bits. To which Block number does byte address $0x4B\_0$ map?

Solution

- 64 blocks' cache ⇒ INDEX = $\log_2(64) = 6$
- Block size of 16 bytes ⇒ OFFSET = $\log_2(16) = 4$
- Address length of 16 bits ⇒ TAG = $16 - 6 - 4 = 6$
- $0x4B\_0 \Rightarrow (4B_0)_{hex} = (0000010010110000)_2$

Exercise 2

The following questions assume a cache whose total size is 1KB and each cache line/block is 64-byte.

1. How many cacheline/blocks does the cache contain?
2. Suppose the cache is a direct mapped cache. Given a 32-bit address 0xAB345f78, which cache-line/block contains the cached data for this address?
3. How many cacheline-aligned memory addresses can be mapped to the same cache location as 0xAB345f78 (assuming 32-bit address space)?
4. If your answer above. is bigger than 1, how can we determine which of the memory addresses is actually stored at a given cache location?

Solution

1. 1KB÷64byte = $2^{10} \div 2^6 = 2^4 = 16$ cache lines/blocks.
2. INDEX = log(16) = 4 bits; OFFSET = log(64) = 6 bits; TAG = $32 - 4 - 6 = 22$ bits. Thus the index is the 23rd to 26th bit of location, which is 1101(13th line of cache).
3. Since the TAG has 22 bits, the answer is 2^{22}.
4. We should compare the TAG.

Exercise 3

How many total bits are required for a direct-mapped cache with 16KB of data and 16 bytes block size, assuming 32-bit address?

Solution

- OFFSET: 16 bytes block size ⇒ OFFSET = log(16) = 4 bits
- INDEX: $16KB \div 16bytes = 2^{10}$ lines ⇒ INDEX = 10 bits
- TAG: ADDRESS = TAG + INDEX + OFFSET ⇒ TAG = $32 - 4 - 10 = 18$ bits
- TotalBitsInCache = (BlockSizeInBits + V-bit + TAG)*(No.OfBlocks) ⇒ $(128 + 1 + 18) * 2^{10} =$ 147kbits

9.3 Associative Cache

1. 2-way Associative Cache
2. N-way Associative Cache

The size of cache is calculated as:

CacheSize = #Sets * #Ways * TotalBlockSize,
TotalBlockSize = BlockSizeInBits + TAG + V-bit + w*R-bit + v*D-bit
w = 1 if more than 1 way (otherwise 0), v = 1 if write-back (otherwise 0)

Exercise

Consider a 64 Blocks 2-way associative cache and a block size of 16 bytes. To what Block number does byte address $(0x4B_0)_{16} = (0000010010110000)_2$ map to (assuming 16 bits address)?

Solution

- 2-way associative 64 blocks ⇒ 32 lines
- INDEX: 32 lines ⇒ 5 bits
- OFFSET: 16 bytes ⇒ 4 bits
- TAG: 16 bits address ⇒ $16 - 5 - 4 = 7$ bites

9.4 Performance

Cache Performance Computing Formula:

$$\text{AMAT} = \text{HitTime} + \text{MissRate} * \text{MissPenalty}$$

- AMAT: Average Memory Accessing Time
- HitTime: the time it takes to retrieve data
- MissRate: #CacheMisses ÷ #CacheAccesses
- MissPenalty: the additional time spent when a cache miss happens

As we increase the number of "ways" in the cache, the **HitTime** will increase because CPU needs to scan through the content in one line cache ; the **MissRate** will decrease because each line of cache will contain more content.

To achieve the best performance, we should follow these rules to optimize AMAT:

- To lower HitTime: Small + Simple + Low Associativity
- To lower MissRate: Large Block Size + Large Cache + High Associativity
- To lower MissPenalty: Small Block Size + Multi-Level

9.5 Writing Policy

There are different ways for cache to react under the situation of cache Hit or Miss. For write hit, we have **Write Through Cache** and **Write Back Cache**:

1. **Write Through Cache:** has additional Write Buffer
   - good: Simple code logic
   - good: Cache and memory consistent
   - bad: Many writes to memory
2. **Write Back Cache:** has dirty in cache line
   - good: Fewer writes to memory
   - bad: Cache and memory inconsistent
   - bad: Requires dirty bits

As for the situation of write miss, we have **Write allocate (fetch then write)** and **No-Write allocate (write around)**.

1. **Write allocate**
   - good: Cache and memory consistent
   - bad: More writes
2. **No-Write allocate**
   - good: Fewer writes
   - bad: Cache and memory inconsistent

9.6 Summary

Cache Architecture

- Principles:
  - Spatial Locality
  - Temporal Locality
- Cache Size
  - #Blocks
  - tag, Vbit, Dbit, Rbit
- Mapping
  - Direct Mapping
  - 2-way Associative
  - ...
  - Fully Associative
- Replazement Methods
  - Random
  - FIFO
  - LRU
  - NRU (Rbit)
  - Priority with (RVD)bits: v=0 → R=0&D=0 → R=0&D=1 → R=1&D=0 → Randomly
- Write Polity
  - Write Hit
    - write-through ← write to the next memory and cache
    - write-back ← write to cache only (Dbit), write to next memory when replace
  - Write Miss
    - write-allocate ← fetch, then write
    - write-around ← write to next memory, then fetch
- AMAT = HitTime + MissRate * MissPenalty

9.7 More Exercises

TO BE COMPLETED (please refer to Lecture/Lab slides on BS)

10 Digital Logic

10.1 Transistor and Basic Gates

What is a Transistor?

- Current controlled switch;
- Three layers of semi-conductor materials;
- A small flow of electrons from emitter to the base will turn the on and allow a large flow of electrons from the emitter to the collector.

Types of Transistors

1. n-type (NPN)
    - when Gate has positive voltage, switch closed;
    - when Gate has zero voltage, switch opened.
2. p-type (PNP)
    - when Gate has positive voltage, switch opened;
    - when Gate has zero voltage, switch closed.

Logic Gates

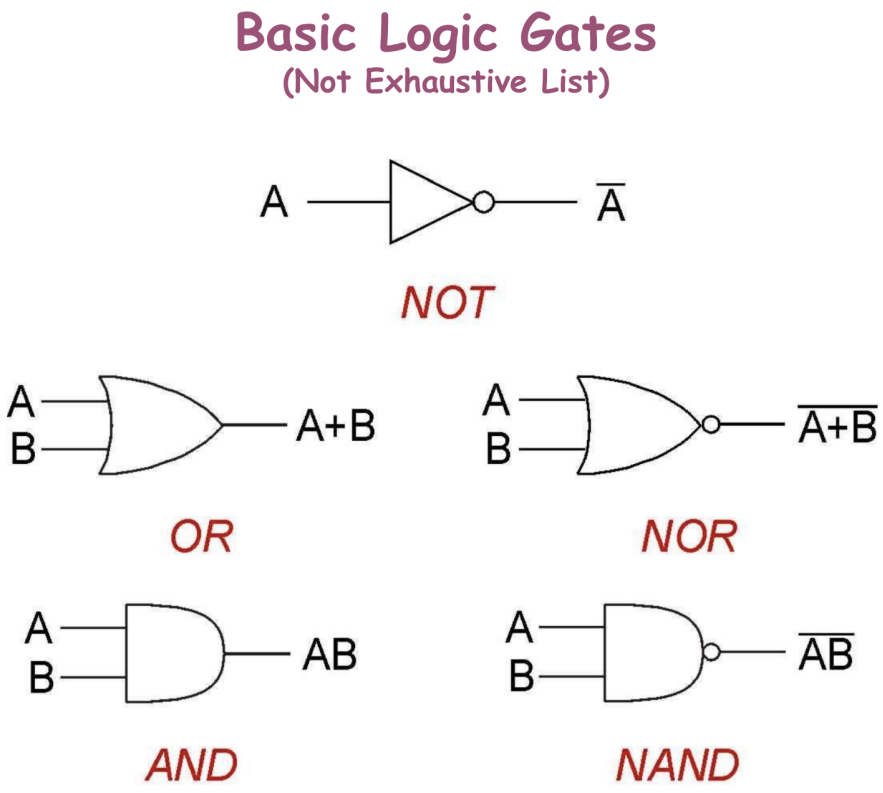

Figure 1: 10.1 Basic Gates (from lecture slides)

A	B	A AND B	A OR B	NOT A	A XOR B	A NAND B
0	0	0	0	1	0	1
0	1	0	1	1	1	1
1	0	0	1	0	1	1
1	1	1	1	0	0	0

Table 8: Truth table for basic logic gates

10.2 Circuits and 1-bit Adder

Once we have the transistors, we can build basic logic gates; once we have the gates, we can start to build circuits.

How to Build a Circuit?

- Logistics:
  1. Look at truth table: look at rows with output=1 and use (+);
  2. Derive the logic function: simplify the equation using the rules of boolean algebra;
  3. Build the circuit: use basic gates
- Rules:
  1. Identity Law: $A + 0 = A$ and $A \cdot 1 = A$
  2. Null Law (Annulment Law): $A + A' = 1$ and $A \cdot A' = 0$
  3. Domination Law: $A + 1 = 1$ and $A \cdot 0 = 0$
  4. Idempotent Law: $A + A = A$ and $A \cdot A = A$
  5. Complement Law: $A + A' = 1$ and $A \cdot A' = 0$
  6. Commutative Law: $A + B = B + A$ and $A \cdot B = B \cdot A$
  7. Associative Law: $A + (B + C) = (A + B) + C$ and $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
  8. Distributive Law: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ and $A + (B \cdot C) = (A + B) \cdot (A + C)$
  9. De Morgan's Theorem: $(A + B)' = A' \cdot B'$ and $(A \cdot B)' = A' + B'$
  10. Absorption Law: $A + (A \cdot B) = A$ and $A \cdot (A + B) = A$

1-bit Adder

Knowing the above rules, let's start with something simple: 1-bit adder. First we need the truth table:

A	B	Cin	$Cout$	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 9: Truth Table for 1-bit Adder

Then we should derive the boolean equation(logic function) from it. See below:

$$C_{out} = \bar{A} \cdot B \cdot C_{in} + A \cdot \bar{B} \cdot C_{in} + A \cdot B \cdot \bar{C_{in}} + A \cdot B \cdot C_{in}$$
$$S = \bar{A} \cdot \bar{B} \cdot C_{in} + \bar{A} \cdot B \cdot \bar{C_{in}} + A \cdot \bar{B} \cdot \bar{C_{in}} + A \cdot B \cdot C_{in}$$

10.3 Important Circuits

2-bit Multiplication

A_1	A_0	B_1	B_0	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
1	1	1	1	1	0	0	1

Table 10: Truth Table for 2-bit Multiplication

2-to-4 Decoder

I_1	I_0	$D3$	$D2$	$D1$	$D0$
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Table 11: Truth Table for 2-to-4 Decoder

The given input (in Binary) makes the i^{th} output one.

$$D0 = \bar{I_1} \cdot \bar{I_0}; D1 = \bar{I_1} \cdot I_0; D2 = I_1 \cdot \bar{I_0}; D3 = I_1 \cdot I_0.$$

4-to-1 Multiplexer

S_1	S_0	I_3	I_2	I_1	I_0	O
0	0	x	x	x	0	0
0	1	x	x	1	x	1
1	0	x	0	x	x	0
1	1	1	x	x	x	1

Table 12: Truth Table for 4-to-1 Multiplexer

Using the select lines, you can reflect one of the inputs to the output.

$$O = (\bar{S_1} \cdot \bar{S_0} \cdot I_0) + (\bar{S_1} \cdot S_0 \cdot I_1) + (S_1 \cdot \bar{S_0} \cdot I_2) + (S_1 \cdot S_0 \cdot I_3)$$

Summary

1. **Adder/Subtractor**: Adders and subtractors are fundamental in digital circuits, used for performing basic arithmetic operations. Adders sum binary numbers, while subtractors perform subtraction. They are crucial in arithmetic logic units (ALUs), CPUs, and various digital systems. The simplicity of their logic allows them to be cascaded to handle binary numbers of any size. Their efficiency and speed are vital for the overall performance of computational systems.

2. **Multiplier**: Multipliers are used to perform multiplication operations in digital systems. They are more complex than adders, requiring a larger number of gates and more intricate logic. Multipliers are essential in various computing tasks, especially in graphics, scientific computations, and signal processing. They can be designed using various methods, such as the shift-and-add method, and can significantly impact the processing speed of a system.

3. **Decoder**: Decoders are used to convert coded inputs into coded outputs, often from binary to a more usable form. They are widely used in memory address decoding, enabling the selection of specific memory addresses. Decoders are also used in display systems, such as 7-segment displays, to convert binary inputs into human-readable form. They are fundamental in digital systems for data routing and selection.

4. **Multiplexer**: Multiplexers, or muxes, select one line of input from multiple inputs and direct it to a single output line. They are crucial in digital circuits for data selection and routing, allowing for the efficient use of data paths. Multiplexers are widely used in communication systems for signal multiplexing and in CPUs for routing data to different parts of the system.

5. **Arithmetic Logic Unit (ALU)**: The ALU is a core component of CPUs and microprocessors, responsible for performing arithmetic and logic operations. It combines the functionalities of adders, subtractors, and other logic circuits to execute a wide range of operations. The ALU is critical in processing the mathematical and logical tasks of a computer, influencing the processor's performance and efficiency.

10.4 Sequential Circuits

A sequential circuit is a type of digital circuit that includes storage elements, allowing it to store and process information. Unlike combinational circuits, sequential circuits depend not only on the current inputs but also on previous inputs and states. There are two main types of sequential circuits: latches and flip-flops.

Latch

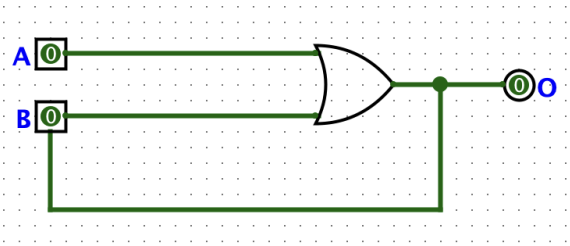

Figure 2: 10.4 Simple Latch

A latch is a basic sequential circuit element commonly used to store a single binary data bit. It can store information and update its stored state based on changes in input signals.

SR-Latch

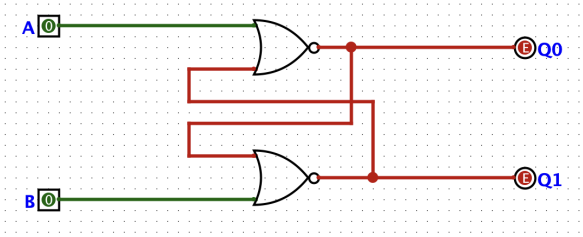

Figure 3: 10.4 SR Latch

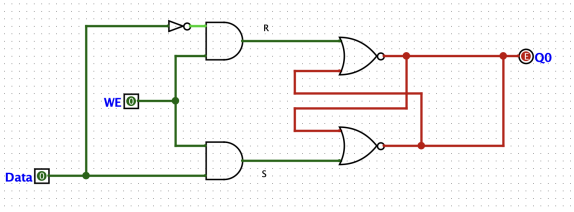

Figure 4: 10.4 Gated SR Latch

Latches come in several types, and one common type is the SR latch (Set-Reset Latch). An SR latch has two inputs, S (Set), and R (Reset), as well as one output. When the S input is set to 1 and the R input is set to 0, the latch is in the set state, and its output is 1. When the S input is 0 and the R input is 1, the latch is in the reset state, and its output is 0. When both S and R are 0, the latch maintains its current state. SR latches can be used to build more complex sequential circuits such as registers and counters.

Once we have gated latch, we can now move on to a key component: Flip-Flop.

Flip-Flop

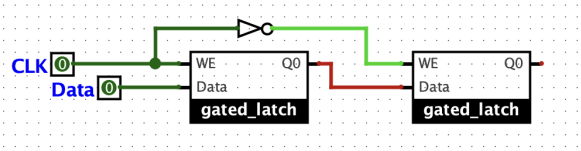

Figure 5: 10.4 Flip-Flop

A flip-flop is a digital circuit element used for storing binary information in sequential logic circuits. It has two stable states, typically represented as '0' and '1', and can transition between these states based on clock signals and input data. Flip-flops are fundamental building blocks in digital electronics and are widely used for memory storage and state retention in electronic devices and digital systems.

The primary purpose of sequential circuits is to store and process information to perform more complex digital functions such as storage, counting, and timing control. Latches and flip-flops are fundamental building blocks for constructing these sequential circuits, enabling the storage and transfer of information.

SR Latch (in detail)

- **Set:** $S = 1, R = 0 \Rightarrow Q = 1$;
- **Reset:** $S = 0, R = 1 \Rightarrow Q = 0$;
- **No Change:** $S = 0, R = 0 \Rightarrow$ No change;
- **Error:** $S = 1, R = 1 \Rightarrow$ Meaningless.

Gated SR Latch (in detail)

- **Set(SR):** $Data = 1, WE = 1 \Rightarrow [S = 1, R = 0] \Rightarrow Q = 1$;
- **Reset(SR):** $Data = 0, WE = 1 \Rightarrow [S = 0, R = 1] \Rightarrow Q = 0$.

Flip-Flop (in detail)

- CLK=1 $\Rightarrow$ Data will be stored in first latch only;
- CLK=0 $\Rightarrow$ First latch ontput will be stored in the second.

1. Ensure correct output from the circuit;
2. Helps synchronizing the operations.

10.5 Register File

Byte (see Figure 6)

A byte is a unit of data storage in computing, typically consisting of 8 bits. It is a fundamental building block for representing and processing data, and it can represent a wide range of values from 0 to 255 in binary.

Register (see Figure 7)

A register is a small, high-speed storage location within a CPU (Central Processing Unit) used for temporary data storage and manipulation during program execution. Registers are crucial for performing arithmetic and logical operations in computer programs.

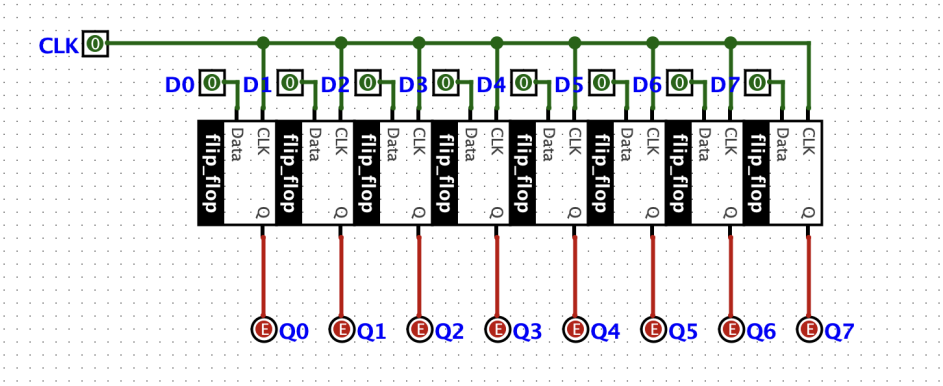

Figure 6: 10.5 Byte

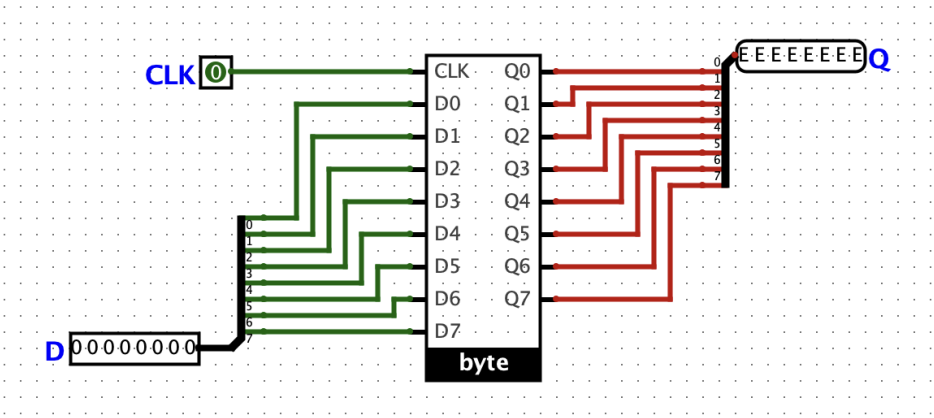

Figure 7: 10.5 Register

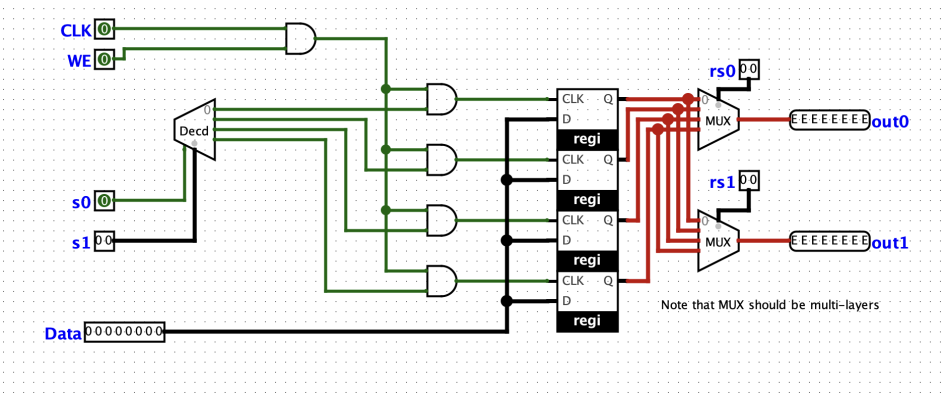

Figure 8: 10.5 Register File

Register File (see Figure 8)

A register file is a collection of multiple registers within a CPU, organized in an array-like structure. It allows the CPU to quickly access and store data for various operations, including arithmetic, logic, and data movement instructions. Register files play a crucial role in computer architecture and microprocessor design.

1. **Decoder:** use decoder to select the register to write by ANDing the decoder output with Write-Enable line and sending result to clock input of corresponding register;
2. **Data:** send writing Data as input to every register;
3. **Output:** the output of all registers should be sent to both multiplexer sets;
4. **MUX:** use two groups of MUX to select the two registers to read from.

10.6 Data Path, Control Units, and DRAM

Data Path

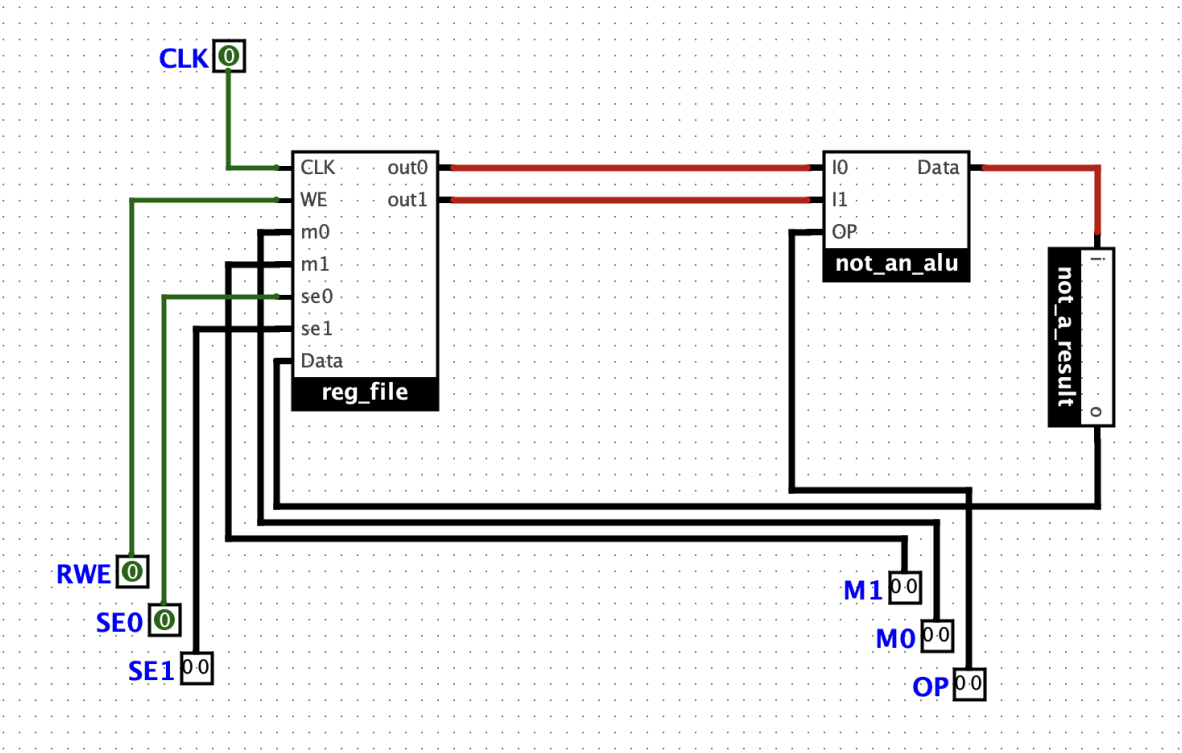

Figure 9: 10.6 Data Path

As we can see from the picture above, mainly there are 6 inputs (RWE, SE0, SE1, M0, M1, OP) and other sources(not in the figure). When we give control signals, they actually corresponds to the above entries. See the example below:

```
//Let SE0=@1, SE1=@2
//Let's say we have a line of Assembly instruction:
add %rax, %rbx  //rbx = rax + rbx
/***
Step 1: @1=rax, @2=rbs, RWE=read, M1&M2=register file, OP=add;
Step 2: result=rax+rbx, Data=result, RWE=write, @1=rbx
E.g.:   RWE @1  @2  other   M0  M1  OP
        0   000 001 ------  00  00  10
***/
```

Control Unit (CU)

- **Function:** The Control Unit orchestrates operations within the CPU, managing data flow and interpreting instructions from memory.
- **Importance:** Essential for executing instructions correctly and efficiently, directly impacting CPU performance and processing speed.
- **Usage:** Manages task sequencing, data flow, and resource allocation within the CPU, using hardwired logic or microprogramming.
- **Write/Read Process:**
  1. Read instruction at (RIP) from next memory;
  2. Decode instruction;
  3. Execute the instruction;
  4. Update (RIP).

Dynamic Random Access Memory (DRAM)

- **Function:** A type of volatile memory in computers, storing data in integrated circuit capacitors, requiring periodic refreshing.
- **Importance:** Provides essential temporary data storage for running applications, influencing overall system performance.
- **Usage:** Main system memory in computers and servers; in graphics cards, variants like GDDR are used for high-speed graphics processing.
- **Write Process:**
  1. Assert word line to HIGH; Apply HIGH charge on bit line. $\rightarrow$ 1;
  2. Assert word line to HIGH; Apply LOW charge on bit line. $\rightarrow$ 0.
- **Read Process:**
  1. Assert word line to HIGH;
  2. Apply MEDIUM charge on bit line.

  if (1) was stored $\rightarrow$ (1) will appear on bit line;
  if (0) was stored $\rightarrow$ (0) will appear on bit line.

Finally...

Thanks Professor Hasan Aljabbouli for the wonderful lectures.